Task-based Asynchronous Pattern (TAP)

.NET Framework (current version)

The Task-based Asynchronous Pattern (TAP) is based on the System.Threading.Tasks.Task and System.Threading.Tasks.Task(Of TResult) types in the System.Threading.Tasks namespace, which are used to represent arbitrary asynchronous operations. TAP is the recommended asynchronous design pattern for new development.

Naming, Parameters, and Return Types

TAP uses a single method to represent the initiation and completion of an asynchronous operation. This is in contrast to the Asynchronous Programming Model (APM or **IAsyncResult**) pattern, which requires `Begin` and `End` methods, and in contrast to the Event-based Asynchronous Pattern (EAP), which requires a method that has the `Async` suffix and also requires one or more events, event handler delegate types, and **EventArg**-derived types. Asynchronous methods in TAP include the `Async` suffix after the operation name; for example, `GetAsync` for a get operation. If you're adding a TAP method to a class that already contains that method name with the `Async` suffix, use the suffix `TaskAsync` instead. For example, if the class already has a `GetAsync` method, use the name `GetTaskAsync`.

The TAP method returns either a System.Threading.Tasks.Task or a System.Threading.Tasks.Task(Of TResult), based on whether the corresponding synchronous method returns void or a type **TResult**.

The parameters of a TAP method should match the parameters of its synchronous counterpart, and should be provided in the same order. However, *out* and *ref* parameters are exempt from this rule and should be avoided entirely. Any data that would have been returned through an *out* or *ref* parameter should instead be returned as part of the **TResult** returned by Task(Of TResult), and should use a tuple or a custom data structure to accommodate multiple values. Methods that are devoted exclusively to the creation, manipulation, or combination of tasks (where the asynchronous intent of the method is clear in the method name or in the name of the type to which the method belongs) need not follow this naming pattern; such methods are often referred to as *combinators*. Examples of combinators include WhenAll and WhenAny, and are discussed in the Using the Built-in Task-based Combinators section of the article Consuming the Task-based Asynchronous Pattern.

For examples of how the TAP syntax differs from the syntax used in legacy asynchronous programming patterns such as the Asynchronous Programming Model (APM) and the Event-based Asynchronous Pattern (EAP), see Asynchronous Programming Patterns.

Initiating an Asynchronous Operation

An asynchronous method that is based on TAP can do a small amount of work synchronously, such as validating arguments and initiating the asynchronous operation, before it returns the resulting task. Synchronous work should be kept to the minimum so the asynchronous method can return quickly. Reasons for a quick return include the following:

- Asynchronous methods may be invoked from user interface (UI) threads, and any long-running synchronous work could harm the responsiveness of the application.
- Multiple asynchronous methods may be launched concurrently. Therefore, any long-running work in the

synchronous portion of an asynchronous method could delay the initiation of other asynchronous operations, thereby decreasing the benefits of concurrency.

In some cases, the amount of work required to complete the operation is less than the amount of work required to launch the operation asynchronously. Reading from a stream where the read operation can be satisfied by data that is already buffered in memory is an example of such a scenario. In such cases, the operation may complete synchronously, and may return a task that has already been completed.

Exceptions

An asynchronous method should raise an exception to be thrown out of the asynchronous method call only in response to a usage error. Usage errors should never occur in production code. For example, if passing a null reference (**Nothing** in Visual Basic) as one of the method's arguments causes an error state (usually represented by an ArgumentNullException exception), you can modify the calling code to ensure that a null reference is never passed. For all other errors, exceptions that occur when an asynchronous method is running should be assigned to the returned task, even if the asynchronous method happens to complete synchronously before the task is returned. Typically, a task contains at most one exception. However, if the task represents multiple operations (for example, WhenAll), multiple exceptions may be associated with a single task.

Target Environment

When you implement a TAP method, you can determine where asynchronous execution occurs. You may choose to execute the workload on the thread pool, implement it by using asynchronous I/O (without being bound to a thread for the majority of the operation's execution), run it on a specific thread (such as the UI thread), or use any number of potential contexts. A TAP method may even have nothing to execute, and may just return a Task that represents the occurrence of a condition elsewhere in the system (for example, a task that represents data arriving at a queued data structure).The caller of the TAP method may block waiting for the TAP method to complete by synchronously waiting on the resulting task, or may run additional (continuation) code when the asynchronous operation completes. The creator of the continuation code has control over where that code executes. You may create the continuation code either explicitly, through methods on the Task class (for example, ContinueWith) or implicitly, by using language support built on top of continuations (for example, **await** in C#, **Await** in Visual Basic, **AwaitValue** in F#).

Task Status

The Task class provides a life cycle for asynchronous operations, and that cycle is represented by the TaskStatus enumeration. To support corner cases of types that derive from Task and Task(Of TResult), and to support the separation of construction from scheduling, the Task class exposes a Start method. Tasks that are created by the public Task constructors are referred to as *cold tasks*, because they begin their life cycle in the non-scheduled Created state and are scheduled only when Start is called on these instances. All other tasks begin their life cycle in a hot state, which means that the asynchronous operations they represent have already been initiated and their task status is an enumeration value other than TaskStatus.Created. All tasks that are returned from TAP methods must be activated. If a TAP method internally uses a task's constructor to instantiate the task to be returned, the TAP method must call Start on the Task object before returning it. Consumers of a TAP method may safely assume that the returned task is active and should not try to call Start on any Task that is returned from a TAP method. Calling Start on an active task results in an InvalidOperationException exception.

Cancellation (Optional)

In TAP, cancellation is optional for both asynchronous method implementers and asynchronous method consumers. If an operation allows cancellation, it exposes an overload of the asynchronous method that accepts a cancellation token (CancellationToken instance). By convention, the parameter is named *cancellationToken*.

VB

```
Public Function ReadAsync(buffer() As Byte, offset As Integer,
                          count As Integer,
                          cancellationToken As CancellationToken) _
                          As Task
```

The asynchronous operation monitors this token for cancellation requests. If it receives a cancellation request, it may choose to honor that request and cancel the operation. If the cancellation request results in work being ended prematurely, the TAP method returns a task that ends in the Canceled state; there is no available result and no exception is thrown. The Canceled state is considered to be a final (completed) state for a task, along with the Faulted and RanToCompletion states. Therefore, if a task is in the Canceled state, its IsCompleted property returns **true**. When a task completes in the Canceled state, any continuations registered with the task are scheduled or executed, unless a continuation option such as NotOnCanceled was specified to opt out of continuation. Any code that is asynchronously waiting for a canceled task through use of language features continues to run but receives an OperationCanceledException or an exception derived from it. Code that is blocked synchronously waiting on the task through methods such as Wait and WaitAll also continue to run with an exception.

If a cancellation token has requested cancellation before the TAP method that accepts that token is called, the TAP method should return a Canceled task. However, if cancellation is requested while the asynchronous operation is running, the asynchronous operation need not accept the cancellation request. The returned task should end in the Canceled state only if the operation ends as a result of the cancellation request. If cancellation is requested but a result or an exception is still produced, the task should end in the RanToCompletion or Faulted state. For asynchronous methods used by a developer who wants cancellation first and foremost, you don't have to provide an overload that doesn't accept a cancellation token. For methods that cannot be canceled, do not provide overloads that accept a cancellation token; this helps indicate to the caller whether the target method is actually cancelable. Consumer code that does not desire cancellation may call a method that accepts a CancellationToken and provide None as the argument value. None is functionally equivalent to the default CancellationToken.

Progress Reporting (Optional)

Some asynchronous operations benefit from providing progress notifications; these are typically used to update a user interface with information about the progress of the asynchronous operation. In TAP, progress is handled through an IProgress(Of T) interface, which is passed to the asynchronous method as a parameter that is usually named *progress*. Providing the progress interface when the asynchronous method is called helps eliminate race conditions that result from incorrect usage (that is, when event handlers that are incorrectly registered after the operation starts may miss updates). More importantly, the progress interface supports varying implementations of progress, as determined by the consuming code. For example, the consuming code may only care about the latest progress update, or may want to buffer all updates, or may want to invoke an action for each update, or may want to control whether the invocation is marshaled to a particular thread. All these options may be achieved by using a different implementation of the interface, customized to the particular consumer's needs. As with cancellation, TAP implementations should provide an IProgress(Of T) parameter only if the API supports progress notifications. For example, if the `ReadAsync` method discussed earlier in this article is able to report intermediate progress in the form of the number of bytes read thus far, the progress callback could be an IProgress(Of T) interface:

VB

```
Public Function ReadAsync(buffer() As Byte, offset As Integer,
                          count As Integer,
                          progress As IProgress(Of Long)) As Task
```

If a `FindFilesAsync` method returns a list of all files that meet a particular search pattern, the progress callback could provide an estimate of the percentage of work completed as well as the current set of partial results. It could do this either with a tuple:

VB

```
Public Function FindFilesAsync(pattern As String,
                               progress As IProgress(Of Tuple(Of Double,
ReadOnlyCollection(Of List(Of FileInfo))))) _
                               As  Task(Of ReadOnlyCollection(Of FileInfo))
```

or with a data type that is specific to the API:

VB

```
Public Function FindFilesAsync(pattern As String,
                               progress As IProgress(Of FindFilesProgressInfo)) _
                               As Task(Of ReadOnlyCollection(Of FileInfo))
```

In the latter case, the special data type is usually suffixed with `ProgressInfo`.

If TAP implementations provide overloads that accept a *progress* parameter, they must allow the argument to be **null**, in which case no progress will be reported. TAP implementations should report the progress to the Progress(Of T) object synchronously, which enables the asynchronous method to quickly provide progress, and allow the consumer of the progress to determine how and where best to handle the information. For example, the progress instance could choose to marshal callbacks and raise events on a captured synchronization context.

IProgress<T> Implementations

The .NET Framework 4.5 provides a single IProgress(Of T) implementation: Progress(Of T). The Progress(Of T) class is declared as follows:

VB

```
Public Class Progress(Of T) : Inherits IProgress(Of T)
    Public Sub New()
    Public Sub New(handler As Action(Of T))
    Protected Overridable Sub OnReport(value As T)
    Public Event ProgressChanged As EventHandler(Of T>
End Class
```

An instance of Progress(Of T) exposes a ProgressChanged event, which is raised every time the asynchronous operation reports a progress update. The ProgressChanged event is raised on the SynchronizationContext object that was captured when the Progress(Of T) instance was instantiated. If no synchronization context was available, a default context that targets the thread pool is used. Handlers may be registered with this event. A single handler may also be provided to the Progress(Of T) constructor for convenience, and behaves just like an event handler for the ProgressChanged event. Progress updates are raised asynchronously to avoid delaying the asynchronous operation while event handlers are executing. Another IProgress(Of T) implementation could choose to apply different semantics.

Choosing the Overloads to Provide

If a TAP implementation uses both the optional CancellationToken and optional IProgress(Of T) parameters, it could potentially require up to four overloads:

VB

```
Public MethodNameAsync(…) As Task
Public MethodNameAsync(…, cancellationToken As CancellationToken cancellationToken) As
Task
Public MethodNameAsync(…, progress As IProgress(Of T)) As Task
Public MethodNameAsync(…, cancellationToken As CancellationToken,
                       progress As IProgress(Of T)) As Task
```

However, many TAP implementations provide neither cancellation or progress capabilities, so they require a single method:

VB

```
Public MethodNameAsync(…) As Task
```

If a TAP implementation supports either cancellation or progress but not both, it may provide two overloads:

VB

```
Public MethodNameAsync(…) As Task
Public MethodNameAsync(…, cancellationToken As CancellationToken) As Task

' … or …

Public MethodNameAsync(…) As Task
Public MethodNameAsync(…, progress As IProgress(Of T)) As Task
```

If a TAP implementation supports both cancellation and progress, it may expose all four overloads. However, it may provide only the following two:

VB

```
Public MethodNameAsync(…) As Task
Public MethodNameAsync(…, cancellationToken As CancellationToken,
                       progress As IProgress(Of T)) As Task
```

To compensate for the two missing intermediate combinations, developers may pass None or a default CancellationToken for the *cancellationToken* parameter and **null** for the *progress* parameter.

If you expect every usage of the TAP method to support cancellation or progress, you may omit the overloads that don't accept the relevant parameter.

If you decide to expose multiple overloads to make cancellation or progress optional, the overloads that don't support cancellation or progress should behave as if they passed None for cancellation or **null** for progress to the overload that does support these.

Related Topics

Title	Description
Asynchronous Programming Patterns	Introduces the three patterns for performing asynchronous operations: the Task-based Asynchronous Pattern (TAP), the Asynchronous Programming Model (APM), and the Event-based Asynchronous Pattern (EAP).
Implementing the Task-based Asynchronous Pattern	Describes how to implement the Task-based Asynchronous Pattern (TAP) in three ways: by using the C# and Visual Basic compilers in Visual Studio, manually, or through a combination of the compiler and manual methods.
Consuming the Task-based Asynchronous Pattern	Describes how you can use tasks and callbacks to achieve waiting without blocking.
Interop with Other Asynchronous Patterns and	Describes how to use the Task-based Asynchronous Pattern (TAP) to implement the Asynchronous Programming Model (APM) and Event-based Asynchronous Pattern

Types	(EAP).

Implementing the Task-based Asynchronous Pattern

.NET Framework (current version)

You can implement the Task-based Asynchronous Pattern (TAP) in three ways: by using the C# and Visual Basic compilers in Visual Studio, manually, or through a combination of the compiler and manual methods. The following sections discuss each method in detail. You can use the TAP pattern to implement both compute-bound and I/O-bound asynchronous operations; the Workloads section discusses each type of operation.

Generating TAP Methods

Using the Compilers

In Visual Studio 2012 and the .NET Framework 4.5, any method that is attributed with the **async** keyword (**Async** in Visual Basic) is considered an asynchronous method, and the C# and Visual Basic compilers perform the necessary transformations to implement the method asynchronously by using TAP. An asynchronous method should return either a System.Threading.Tasks.Task or a System.Threading.Tasks.Task(Of TResult) object. In the case of the latter, the body of the function should return a **TResult**, and the compiler ensures that this result is made available through the resulting task object. Similarly, any exceptions that go unhandled within the body of the method are marshaled to the output task and cause the resulting task to end in the TaskStatus.Faulted state. The exception is when an OperationCanceledException (or derived type) goes unhandled, in which case the resulting task ends in the TaskStatus.Canceled state.

Generating TAP Methods Manually

You may implement the TAP pattern manually for better control over implementation. The compiler relies on the public surface area exposed from the System.Threading.Tasks namespace and supporting types in the System.Runtime.CompilerServices namespace. To implement the TAP yourself, you create a TaskCompletionSource(Of TResult) object, perform the asynchronous operation, and when it completes, call the SetResult, SetException, or SetCanceled method, or the **Try** version of one of these methods. When you implement a TAP method manually, you must complete the resulting task when the represented asynchronous operation completes. For example:

VB

```
<Extension()>
Public Function ReadTask(stream As Stream, buffer() As Byte,
                         offset As Integer, count As Integer,
                         state As Object) As Task(Of Integer)
    Dim tcs As New TaskCompletionSource(Of Integer)()
    stream.BeginRead(buffer, offset, count, Sub(ar)
               Try
                  tcs.SetResult(stream.EndRead(ar))
               Catch exc As Exception
```

```
                    tcs.SetException(exc)
                End Try
            End Sub, state)
    Return tcs.Task
End Function
```

Hybrid Approach

You may find it useful to implement the TAP pattern manually but to delegate the core logic for the implementation to the compiler. For example, you may want to use the hybrid approach when you want to verify arguments outside a compiler-generated asynchronous method so that exceptions can escape to the method's direct caller rather than being exposed through the System.Threading.Tasks.Task object:

VB

```
Public Function MethodAsync(input As String) As Task(Of Integer)
    If input Is Nothing Then Throw New ArgumentNullException("input")

    Return MethodAsyncInternal(input)
End Function

Private Async Function MethodAsyncInternal(input As String) As Task(Of Integer)

   ' code that uses await goes here

   return value
End Function
```

Another case where such delegation is useful is when you're implementing fast-path optimization and want to return a cached task.

Workloads

You may implement both compute-bound and I/O-bound asynchronous operations as TAP methods. However, when TAP methods are exposed publicly from a library, they should be provided only for workloads that involve I/O-bound operations (they may also involve computation, but should not be purely computational). If a method is purely compute-bound, it should be exposed only as a synchronous implementation; the code that consumes it may then choose whether to wrap an invocation of that synchronous method into a task to offload the work to another thread or to achieve parallelism.

Compute-bound Tasks

The System.Threading.Tasks.Task class is ideally suited for representing computationally intensive operations. By default, it takes advantage of special support within the ThreadPool class to provide efficient execution, and it also provides significant control over when, where, and how asynchronous computations execute.

You can generate compute-bound tasks in the following ways:

- In the .NET Framework 4, use the TaskFactory.StartNew method, which accepts a delegate (typically an Action(Of T) or a Func(Of TResult)) to be executed asynchronously. If you provide an Action(Of T) delegate, the method returns a System.Threading.Tasks.Task object that represents the asynchronous execution of that delegate. If you provide a Func(Of TResult) delegate, the method returns a System.Threading.Tasks.Task(Of TResult) object. Overloads of the StartNew method accept a cancellation token (CancellationToken), task creation options (TaskCreationOptions), and a task scheduler (TaskScheduler), all of which provide fine-grained control over the scheduling and execution of the task. A factory instance that targets the current task scheduler is available as a static property (Factory) of the Task class; for example: `Task.Factory.StartNew(…)`.

- In the .NET Framework 4.5, use the static Task.Run method as a shortcut to TaskFactory.StartNew. You may use Run to easily launch a compute-bound task that targets the thread pool. In the .NET Framework 4.5, this is the preferred mechanism for launching a compute-bound task. Use **StartNew** directly only when you want more fine-grained control over the task.

- Use the constructors of the **Task** type or the **Start** method if you want to generate and schedule the task separately. Public methods must only return tasks that have already been started.

- Use the overloads of the Task.ContinueWith method. This method creates a new task that is scheduled when another task completes. Some of the ContinueWith overloads accept a cancellation token, continuation options, and a task scheduler for better control over the scheduling and execution of the continuation task.

- Use the TaskFactory.ContinueWhenAll and TaskFactory.ContinueWhenAny methods. These methods create a new task that is scheduled when all or any of a supplied set of tasks completes. These methods also provide overloads to control the scheduling and execution of these tasks.

In compute-bound tasks, the system can prevent the execution of a scheduled task if it receives a cancellation request before it starts running the task. As such, if you provide a cancellation token (CancellationToken object), you can pass that token to the asynchronous code that monitors the token. You can also provide the token to one of the previously mentioned methods such as **StartNew** or **Run** so that the **Task** runtime may also monitor the token.

For example, consider an asynchronous method that renders an image. The body of the task can poll the cancellation token so that the code may exit early if a cancellation request arrives during rendering. In addition, if the cancellation request arrives before rendering starts, you'll want to prevent the rendering operation:

VB

```
Friend Function RenderAsync(data As ImageData, cancellationToken As _
                            CancellationToken) As Task(Of Bitmap)
    Return Task.Run( Function()
                        Dim bmp As New Bitmap(data.Width, data.Height)
                        For y As Integer = 0 to data.Height - 1
                           cancellationToken.ThrowIfCancellationRequested()
                           For x As Integer = 0 To data.Width - 1
                             ' render pixel [x,y] into bmp
                           Next
                        Next
                        Return bmp
                     End Function, cancellationToken)
End Function
```

Compute-bound tasks end in a Canceled state if at least one of the following conditions is true:

- A cancellation request arrives through the CancellationToken object, which is provided as an argument to the creation method (for example, **StartNew** or **Run**) before the task transitions to the Running state.
- An OperationCanceledException exception goes unhandled within the body of such a task, that exception contains the same CancellationToken that is passed to the task, and that token shows that cancellation is requested.

If another exception goes unhandled within the body of the task, the task ends in the Faulted state, and any attempts to wait on the task or access its result causes an exception to be thrown.

I/O-bound Tasks

To create a task that should not be directly backed by a thread for the entirety of its execution, use the TaskCompletionSource(Of TResult) type. This type exposes a Task property that returns an associated Task(Of TResult) instance. The life cycle of this task is controlled by TaskCompletionSource(Of TResult) methods such as SetResult, SetException, SetCanceled, and their **TrySet** variants.

Let's say that you want to create a task that will complete after a specified period of time. For example, you may want to delay an activity in the user interface. The System.Threading.Timer class already provides the ability to asynchronously invoke a delegate after a specified period of time, and by using TaskCompletionSource(Of TResult) you can put a Task(Of TResult) front on the timer, for example:

VB

```
Public Function Delay(millisecondsTimeout As Integer) As Task(Of DateTimeOffset)
    Dim tcs As TaskCompletionSource(Of DateTimeOffset) = Nothing
    Dim timer As Timer = Nothing

    timer = New Timer( Sub(obj)
                           timer.Dispose()
                           tcs.TrySetResult(DateTimeOffset.UtcNow)
                       End Sub, Nothing, Timeout.Infinite, Timeout.Infinite)

    tcs = New TaskCompletionSource(Of DateTimeOffset)(timer)
    timer.Change(millisecondsTimeout, Timeout.Infinite)
    Return tcs.Task
End Function
```

Starting with the .NET Framework 4.5, the Task.Delay method is provided for this purpose, and you can use it inside another asynchronous method, for example, to implement an asynchronous polling loop:

VB

```
Public Async Function Poll(url As Uri, cancellationToken As CancellationToken,
                           progress As IProgress(Of Boolean)) As Task
    Do While True
        Await Task.Delay(TimeSpan.FromSeconds(10), cancellationToken)
        Dim success As Boolean = False
```

```
        Try
            await DownloadStringAsync(url)
            success = true
        Catch
           ' ignore errors
        End Try
        progress.Report(success)
    Loop
End Function
```

The TaskCompletionSource(Of TResult) class doesn't have a non-generic counterpart. However, Task(Of TResult) derives from Task, so you can use the generic TaskCompletionSource(Of TResult) object for I/O-bound methods that simply return a task. To do this, you can use a source with a dummy **TResult** (Boolean is a good default choice, but if you're concerned about the user of the Task downcasting it to a Task(Of TResult), you can use a private **TResult** type instead). For example, the `Delay` method in the previous example returns the current time along with the resulting offset (`Task<DateTimeOffset>`). If such a result value is unnecessary, the method could instead be coded as follows (note the change of return type and the change of argument to TrySetResult):

VB

```
Public Function Delay(millisecondsTimeout As Integer) As Task(Of Boolean)
     Dim tcs As TaskCompletionSource(Of Boolean) = Nothing
     Dim timer As Timer = Nothing

     Timer = new Timer( Sub(obj)
                           timer.Dispose()
                           tcs.TrySetResult(True)
                        End Sub, Nothing, Timeout.Infinite, Timeout.Infinite)

     tcs = New TaskCompletionSource(Of Boolean)(timer)
     timer.Change(millisecondsTimeout, Timeout.Infinite)
     Return tcs.Task
End Function
```

Mixed Compute-bound and I/O-bound Tasks

Asynchronous methods are not limited to just compute-bound or I/O-bound operations but may represent a mixture of the two. In fact, multiple asynchronous operations are often combined into larger mixed operations. For example, the `RenderAsync` method in a previous example performed a computationally intensive operation to render an image based on some input `imageData`. This `imageData` could come from a web service that you asynchronously access:

VB

```
Public Async Function DownloadDataAndRenderImageAsync(
             cancellationToken As CancellationToken) As Task(Of Bitmap)
    Dim imageData As ImageData = Await DownloadImageDataAsync(cancellationToken)
    Return Await RenderAsync(imageData, cancellationToken)
End Function
```

This example also demonstrates how a single cancellation token may be threaded through multiple asynchronous

operations. For more information, see the cancellation usage section in Consuming the Task-based Asynchronous Pattern.

See Also

Task-based Asynchronous Pattern (TAP)
Consuming the Task-based Asynchronous Pattern
Interop with Other Asynchronous Patterns and Types

Interop with Other Asynchronous Patterns and Types

.NET Framework (current version)

The .NET Framework 1.0 introduced the IAsyncResult pattern, otherwise known as the Asynchronous Programming Model (APM), or the **Begin/End** pattern. The .NET Framework 2.0 added the Event-based Asynchronous Pattern (EAP). Starting with the .NET Framework 4, the Task-based Asynchronous Pattern (TAP) supersedes both APM and EAP, but provides the ability to easily build migration routines from the earlier patterns.

In this topic:

Tasks and the Asynchronous Programming Model (APM)

From APM to TAP

Because the Asynchronous Programming Model (APM) pattern is very structured, it is quite easy to build a wrapper to expose an APM implementation as a TAP implementation. In fact, the .NET Framework, starting with .NET Framework 4, includes helper routines in the form of FromAsync method overloads to provide this translation.

Consider the Stream class and its BeginRead and EndRead methods, which represent the APM counterpart to the synchronous Read method:

VB

```
Public Function Read(buffer As Byte(), offset As Integer,
                     count As Integer) As Integer
```

VB

```
Public Function BeginRead(buffer As Byte, offset As Integer,
                          count As Integer, callback As AsyncCallback,
                          state As Object) As IAsyncResult
```

VB

```
Public Function EndRead(asyncResult As IAsyncResult) As Integer
```

You can use the TaskFactory(Of TResult).FromAsync method to implement a TAP wrapper for this operation as follows:

VB

```
<Extension()>
Public Function ReadAsync(strm As Stream,
                          buffer As Byte(), offset As Integer,
                          count As Integer) As Task(Of Integer)
    If strm Is Nothing Then
       Throw New ArgumentNullException("stream")
    End If

    Return Task(Of Integer).Factory.FromAsync(AddressOf strm.BeginRead,
                                              AddressOf strm.EndRead, buffer,
                                              offset, count, Nothing)
End Function
```

This implementation is similar to the following:

VB

```
<Extension()>
Public Function ReadAsync(stream As Stream, buffer As Byte(), _
                          offset As Integer, count As Integer) _
                          As Task(Of Integer)
   If stream Is Nothing Then
       Throw New ArgumentNullException("stream")
   End If

   Dim tcs As New TaskCompletionSource(Of Integer)()
   stream.BeginRead(buffer, offset, count,
                    Sub(iar)
                       Try
                          tcs.TrySetResult(stream.EndRead(iar))
                       Catch e As OperationCanceledException
                          tcs.TrySetCanceled()
                       Catch e As Exception
                          tcs.TrySetException(e)
                       End Try
                    End Sub, Nothing)
   Return tcs.Task
End Function
```

From TAP to APM

If your existing infrastructure expects the APM pattern, you'll also want to take a TAP implementation and use it where an APM implementation is expected. Because tasks can be composed and the Task class implements IAsyncResult, you can use a straightforward helper function to do this. The following code uses an extension of the Task(Of TResult) class, but you can use an almost identical function for non-generic tasks.

VB

```
<Extension()>
Public Function AsApm(Of T)(task As Task(Of T),
                            callback As AsyncCallback,
                            state As Object) As IAsyncResult
    If task Is Nothing Then
        Throw New ArgumentNullException("task")
    End If

    Dim tcs As New TaskCompletionSource(Of T)(state)
    task.ContinueWith(Sub(antecedent)
                          If antecedent.IsFaulted Then
                             tcs.TrySetException(antecedent.Exception.InnerExceptions)
                          ElseIf antecedent.IsCanceled Then
                             tcs.TrySetCanceled()
                          Else
                             tcs.TrySetResult(antecedent.Result)
                          End If

                          If callback IsNot Nothing Then
                             callback(tcs.Task)
                          End If
                      End Sub, TaskScheduler.Default)
    Return tcs.Task
End Function
```

Now, consider a case where you have the following TAP implementation:

VB

```
Public Shared Function DownloadStringAsync(url As Uri) As Task(Of String)
```

and you want to provide this APM implementation:

VB

```
Public Function BeginDownloadString(url As Uri,
                                    callback As AsyncCallback,
                                    state As Object) As IAsyncResult
```

VB

```
Public Function EndDownloadString(asyncResult As IAsyncResult) As String
```

The following example demonstrates one migration to APM:

VB

```
Public Function BeginDownloadString(url As Uri,
                                    callback As AsyncCallback,
```

```
                                        state As Object) As IAsyncResult
    Return DownloadStringAsync(url).AsApm(callback, state)
End Function

Public Function EndDownloadString(asyncResult As IAsyncResult) As String
    Return CType(asyncResult, Task(Of String)).Result
End Function
```

Tasks and the Event-based Asynchronous Pattern (EAP)

Wrapping an Event-based Asynchronous Pattern (EAP) implementation is more involved than wrapping an APM pattern, because the EAP pattern has more variation and less structure than the APM pattern. To demonstrate, the following code wraps the `DownloadStringAsync` method. `DownloadStringAsync` accepts a URI, raises the `DownloadProgressChanged` event while downloading in order to report multiple statistics on progress, and raises the `DownloadStringCompleted` event when it's done. The final result is a string that contains the contents of the page at the specified URI.

VB

```
Public Shared Function DownloadStringAsync(url As Uri) As Task(Of String)
    Dim tcs As New TaskCompletionSource(Of String)()
    Dim wc As New WebClient()
    AddHandler wc.DownloadStringCompleted, Sub(s,e)
            If e.Error IsNot Nothing Then
               tcs.TrySetException(e.Error)
            ElseIf e.Cancelled Then
               tcs.TrySetCanceled()
            Else
               tcs.TrySetResult(e.Result)
            End If
        End Sub
    wc.DownloadStringAsync(url)
    Return tcs.Task
End Function
```

Tasks and Wait Handles

From Wait Handles to TAP

Although wait handles don't implement an asynchronous pattern, advanced developers may use the WaitHandle class and the ThreadPool.RegisterWaitForSingleObject method for asynchronous notifications when a wait handle is set. You can wrap the RegisterWaitForSingleObject method to enable a task-based alternative to any synchronous wait on a wait handle:

VB

```
<Extension()>
Public Function WaitOneAsync(waitHandle As WaitHandle) As Task
    If waitHandle Is Nothing Then
        Throw New ArgumentNullException("waitHandle")
    End If

    Dim tcs As New TaskCompletionSource(Of Boolean)()
    Dim rwh As RegisteredWaitHandle =
ThreadPool.RegisterWaitForSingleObject(waitHandle,
        Sub(state, timedOut)
           tcs.TrySetResult(True)
        End Sub, Nothing, -1, True)
    Dim t = tcs.Task
    t.ContinueWith( Sub(antecedent)
                       rwh.Unregister(Nothing)
                   End Sub)
    Return t
End Function
```

With this method, you can use existing WaitHandle implementations in asynchronous methods. For example, if you want to throttle the number of asynchronous operations that are executing at any particular time, you can utilize a semaphore (a System.Threading.SemaphoreSlim object). You can throttle to *N* the number of operations that run concurrently by initializing the semaphore's count to *N*, waiting on the semaphore any time you want to perform an operation, and releasing the semaphore when you're done with an operation:

VB

```
Shared N As Integer = 3

Shared m_throttle As New SemaphoreSlim(N, N)

Shared Async Function DoOperation() As Task
    Await m_throttle.WaitAsync()
    ' Do work.
    m_throttle.Release()
End Function
```

You can also build an asynchronous semaphore that does not rely on wait handles and instead works completely with tasks. To do this, you can use techniques such as those discussed in Consuming the Task-based Asynchronous Pattern for building data structures on top of Task.

From TAP to Wait Handles

As previously mentioned, the Task class implements IAsyncResult, and that implementation exposes an IAsyncResult.AsyncWaitHandle property that returns a wait handle that will be set when the Task completes. You can get a WaitHandle for a Task as follows:

VB

```
Dim wh As WaitHandle = CType(task, IAsyncResult).AsyncWaitHandle
```

See Also

Task-based Asynchronous Pattern (TAP)
Implementing the Task-based Asynchronous Pattern
Consuming the Task-based Asynchronous Pattern

Consuming the Task-based Asynchronous Pattern

.NET Framework (current version)

When you use the Task-based Asynchronous Pattern (TAP) to work with asynchronous operations, you can use callbacks to achieve waiting without blocking. For tasks, this is achieved through methods such as Task.ContinueWith. Language-based asynchronous support hides callbacks by allowing asynchronous operations to be awaited within normal control flow, and compiler-generated code provides this same API-level support.

Suspending Execution with Await

Starting with the .NET Framework 4.5, you can use the await (C# Reference) keyword in C# and the Await Operator (Visual Basic) in Visual Basic to asynchronously await Task and Task(Of TResult) objects. When you're awaiting a Task, the **await** expression is of type **void**. When you're awaiting a Task(Of TResult), the **await** expression is of type **TResult**. An **await** expression must occur inside the body of an asynchronous method. For more information about C# and Visual Basic language support in the .NET Framework 4.5, see the C# and Visual Basic language specifications.

Under the covers, the await functionality installs a callback on the task by using a continuation. This callback resumes the asynchronous method at the point of suspension. When the asynchronous method is resumed, if the awaited operation completed successfully and was a Task(Of TResult), its **TResult** is returned. If the Task or Task(Of TResult) that was awaited ended in the Canceled state, an OperationCanceledException exception is thrown. If the Task or Task(Of TResult) that was awaited ended in the Faulted state, the exception that caused it to fault is thrown. A **Task** can fault as a result of multiple exceptions, but only one of these exceptions is propagated. However, the Task.Exception property returns an AggregateException exception that contains all the errors.

If a synchronization context (SynchronizationContext object) is associated with the thread that was executing the asynchronous method at the time of suspension (for example, if the SynchronizationContext.Current property is not **null**), the asynchronous method resumes on that same synchronization context by using the context's Post method. Otherwise, it relies on the task scheduler (TaskScheduler object) that was current at the time of suspension. Typically, this is the default task scheduler (TaskScheduler.Default), which targets the thread pool. This task scheduler determines whether the awaited asynchronous operation should resume where it completed or whether the resumption should be scheduled. The default scheduler typically allows the continuation to run on the thread that the awaited operation completed.

When an asynchronous method is called, it synchronously executes the body of the function up until the first await expression on an awaitable instance that has not yet completed, at which point the invocation returns to the caller. If the asynchronous method does not return **void**, a Task or Task(Of TResult) object is returned to represent the ongoing computation. In a non-void asynchronous method, if a return statement is encountered or the end of the method body is reached, the task is completed in the RanToCompletion final state. If an unhandled exception causes control to leave the body of the asynchronous method, the task ends in the Faulted state. If that exception is an OperationCanceledException, the task instead ends in the Canceled state. In this manner, the result or exception is eventually published.

There are several important variations of this behavior. For performance reasons, if a task has already completed by the time the task is awaited, control is not yielded, and the function continues to execute. Additionally, returning to the original context isn't always the desired behavior and can be changed; this is described in more detail in the next section.

Configuring Suspension and Resumption with Yield and ConfigureAwait

Several methods provide more control over an asynchronous method's execution. For example, you can use the Task.Yield method to introduce a yield point into the asynchronous method:

C#

```
public class Task : …
{
    public static YieldAwaitable Yield();
    …
}
```

This is equivalent to asynchronously posting or scheduling back to the current context.

C#

```
Task.Run(async  delegate
{
    for(int i=0; i<1000000; i++)
    {
        await Task.Yield(); // fork the continuation into a separate work item
        ...
    }
});
```

You can also use the Task.ConfigureAwait method for better control over suspension and resumption in an asynchronous method. As mentioned previously, by default, the current context is captured at the time an asynchronous method is suspended, and that captured context is used to invoke the asynchronous method's continuation upon resumption. In many cases, this is the exact behavior you want. In other cases, you may not care about the continuation context, and you can achieve better performance by avoiding such posts back to the original context. To enable this, use the Task.ConfigureAwait method to inform the await operation not to capture and resume on the context, but to continue execution wherever the asynchronous operation that was being awaited completed:

C#

```
await someTask.ConfigureAwait(continueOnCapturedContext:false);
```

Canceling an Asynchronous Operation

Starting with the .NET Framework 4, TAP methods that support cancellation provide at least one overload that accepts a cancellation token (CancellationToken object).

A cancellation token is created through a cancellation token source (CancellationTokenSource object). The source's Token

property returns the cancellation token that will be signaled when the source's Cancel method is called. For example, if you want to download a single webpage and you want to be able to cancel the operation, you create a CancellationTokenSource object, pass its token to the TAP method, and then call the source's Cancel method when you're ready to cancel the operation:

C#

```
var cts = new CancellationTokenSource();
string result = await DownloadStringAsync(url, cts.Token);
… // at some point later, potentially on another thread
cts.Cancel();
```

To cancel multiple asynchronous invocations, you can pass the same token to all invocations:

C#

```
var cts = new CancellationTokenSource();
    IList<string> results = await Task.WhenAll(from url in urls select DownloadStringAsync
    // at some point later, potentially on another thread
    …
    cts.Cancel();
```

Or, you can pass the same token to a selective subset of operations:

C#

```
var cts = new CancellationTokenSource();
    byte [] data = await DownloadDataAsync(url, cts.Token);
    await SaveToDiskAsync(outputPath, data, CancellationToken.None);
    … // at some point later, potentially on another thread
    cts.Cancel();
```

Cancellation requests may be initiated from any thread.

You can pass the CancellationToken.None value to any method that accepts a cancellation token to indicate that cancellation will never be requested. This causes the CancellationToken.CanBeCanceled property to return **false**, and the called method can optimize accordingly. For testing purposes, you can also pass in a pre-canceled cancellation token that is instantiated by using the constructor that accepts a Boolean value to indicate whether the token should start in an already-canceled or not-cancelable state.

This approach to cancellation has several advantages:

- You can pass the same cancellation token to any number of asynchronous and synchronous operations.
- The same cancellation request may be proliferated to any number of listeners.
- The developer of the asynchronous API is in complete control of whether cancellation may be requested and when it may take effect.

- The code that consumes the API may selectively determine the asynchronous invocations that cancellation requests will be propagated to.

Monitoring Progress

Some asynchronous methods expose progress through a progress interface passed into the asynchronous method. For example, consider a function which asynchronously downloads a string of text, and along the way raises progress updates that include the percentage of the download that has completed thus far. Such a method could be consumed in a Windows Presentation Foundation (WPF) application as follows:

C#

```
private async  void btnDownload_Click(object sender, RoutedEventArgs e)
{
    btnDownload.IsEnabled = false;
    try
    {
        txtResult.Text = await DownloadStringAsync(txtUrl.Text,
            new Progress<int>(p => pbDownloadProgress.Value = p));
    }
    finally { btnDownload.IsEnabled = true; }
}
```

Using the Built-in Task-based Combinators

The System.Threading.Tasks namespace includes several methods for composing and working with tasks.

Task.Run

The Task class includes several Run methods that let you easily offload work as a Task or Task(Of TResult) to the thread pool, for example:

C#

```
public async  void button1_Click(object sender, EventArgs e)
{
    textBox1.Text = await Task.Run(() =>
    {
        // … do compute-bound work here
        return answer;
    });
}
```

Some of these Run methods, such as the Task.Run(Func(Of Task)) overload, exist as shorthand for the TaskFactory.StartNew method. Other overloads, such as Task.Run(Func(Of Task)), enable you to use await within the offloaded work, for example:

C#

```
public async  void button1_Click(object sender, EventArgs e)
{
    pictureBox1.Image = await Task.Run(async() =>
    {
        using(Bitmap bmp1 = await DownloadFirstImageAsync())
        using(Bitmap bmp2 = await DownloadSecondImageAsync())
        return Mashup(bmp1, bmp2);
    });
}
```

Such overloads are logically equivalent to using the TaskFactory.StartNew method in conjunction with the Unwrap extension method in the Task Parallel Library.

Task.FromResult

Use the FromResult(Of TResult) method in scenarios where data may already be available and just needs to be returned from a task-returning method lifted into a Task(Of TResult):

C#

```
public Task<int> GetValueAsync(string key)
{
    int cachedValue;
    return TryGetCachedValue(out cachedValue) ?
        Task.FromResult(cachedValue) :
        GetValueAsyncInternal();
}

private async  Task<int> GetValueAsyncInternal(string key)
{
    …
}
```

Task.WhenAll

Use the WhenAll method to asynchronously wait on multiple asynchronous operations that are represented as tasks. The method has multiple overloads that support a set of non-generic tasks or a non-uniform set of generic tasks (for example, asynchronously waiting for multiple void-returning operations, or asynchronously waiting for multiple value-returning methods where each value may have a different type) and to support a uniform set of generic tasks (such as asynchronously waiting for multiple **TResult**-returning methods).

Let's say you want to send email messages to several customers. You can overlap sending the messages so you're not

waiting for one message to complete before sending the next. You can also find out when the send operations have completed and whether any errors have occurred:

C#

```
IEnumerable<Task> asyncOps = from addr in addrs select SendMailAsync(addr);
await Task.WhenAll(asyncOps);
```

This code doesn't explicitly handle exceptions that may occur, but lets exceptions propagate out of the **await** on the resulting task from WhenAll. To handle the exceptions, you can use code such as the following:

C#

```
IEnumerable<Task> asyncOps = from addr in addrs select SendMailAsync(addr);
try
{
    await Task.WhenAll(asyncOps);
}
catch(Exception exc)
{
    ...
}
```

In this case, if any asynchronous operation fails, all the exceptions will be consolidated in an AggregateException exception, which is stored in the Task that is returned from the WhenAll method. However, only one of those exceptions is propagated by the **await** keyword. If you want to examine all the exceptions, you can rewrite the previous code as follows:

C#

```
Task [] asyncOps = (from addr in addrs select SendMailAsync(addr)).ToArray();
try
{
    await Task.WhenAll(asyncOps);
}
catch(Exception exc)
{
    foreach(Task faulted in asyncOps.Where(t => t.IsFaulted))
    {
        … // work with faulted and faulted.Exception
    }
}
```

Let's consider an example of downloading multiple files from the web asynchronously. In this case, all the asynchronous operations have homogeneous result types, and it's easy to access the results:

C#

```
string [] pages = await Task.WhenAll(
```

```
    from url in urls select DownloadStringAsync(url));
```

You can use the same exception-handling techniques we discussed in the previous void-returning scenario:

C#

```
Task [] asyncOps =
    (from url in urls select DownloadStringAsync(url)).ToArray();
try
{
    string [] pages = await Task.WhenAll(asyncOps);
    ...
}
catch(Exception exc)
{
    foreach(Task<string> faulted in asyncOps.Where(t => t.IsFaulted))
    {
        … // work with faulted and faulted.Exception
    }
}
```

Task.WhenAny

You can use the WhenAny method to asynchronously wait for just one of multiple asynchronous operations represented as tasks to complete. This method serves four primary use cases:

- Redundancy: Performing an operation multiple times and selecting the one that completes first (for example, contacting multiple stock quote web services that will produce a single result and selecting the one that completes the fastest).
- Interleaving: Launching multiple operations and waiting for all of them to complete, but processing them as they complete.
- Throttling: Allowing additional operations to begin as others complete. This is an extension of the interleaving scenario.
- Early bailout: For example, an operation represented by task t1 can be grouped in a WhenAny task with another task t2, and you can wait on the WhenAny task. Task t2 could represent a time-out, or cancellation, or some other signal that causes the WhenAny task to complete before t1 completes.

Redundancy

Consider a case where you want to make a decision about whether to buy a stock. There are several stock recommendation web services that you trust, but depending on daily load, each service can end up being slow at different times. You can use the WhenAny method to receive a notification when any operation completes:

C#

```
var recommendations = new List<Task<bool>>()
{
    GetBuyRecommendation1Async(symbol),
    GetBuyRecommendation2Async(symbol),
    GetBuyRecommendation3Async(symbol)
};
Task<bool> recommendation = await Task.WhenAny(recommendations);
if (await recommendation) BuyStock(symbol);
```

Unlike WhenAll, which returns the unwrapped results of all tasks that completed successfully, WhenAny returns the task that completed. If a task fails, it's important to know that it failed, and if a task succeeds, it's important to know which task the return value is associated with. Therefore, you need to access the result of the returned task, or further await it, as this example shows.

As with WhenAll, you have to be able to accommodate exceptions. Because you receive the completed task back, you can await the returned task to have errors propagated, and **try/catch** them appropriately; for example:

C#

```
Task<bool> [] recommendations = …;
while(recommendations.Count > 0)
{
    Task<bool> recommendation = await Task.WhenAny(recommendations);
    try
    {
        if (await recommendation) BuyStock(symbol);
        break;
    }
    catch(WebException exc)
    {
        recommendations.Remove(recommendation);
    }
}
```

Additionally, even if a first task completes successfully, subsequent tasks may fail. At this point, you have several options for dealing with exceptions: You can wait until all the launched tasks have completed, in which case you can use the WhenAll method, or you can decide that all exceptions are important and must be logged. For this, you can use continuations to receive a notification when tasks have completed asynchronously:

C#

```
foreach(Task recommendation in recommendations)
{
    var ignored = recommendation.ContinueWith(
        t => { if (t.IsFaulted) Log(t.Exception); });
}
```

or:

C#

```
foreach(Task recommendation in recommendations)
{
    var ignored = recommendation.ContinueWith(
        t => Log(t.Exception), TaskContinuationOptions.OnlyOnFaulted);
}
```

or even:

```
private static async  void LogCompletionIfFailed(IEnumerable<Task> tasks)
{
    foreach(var task in tasks)
    {
        try { await task; }
        catch(Exception exc) { Log(exc); }
    }
}
…
LogCompletionIfFailed(recommendations);
```

Finally, you may want to cancel all the remaining operations:

C#

```
var cts = new CancellationTokenSource();
var recommendations = new List<Task<bool>>()
{
    GetBuyRecommendation1Async(symbol, cts.Token),
    GetBuyRecommendation2Async(symbol, cts.Token),
    GetBuyRecommendation3Async(symbol, cts.Token)
};

Task<bool> recommendation = await Task.WhenAny(recommendations);
cts.Cancel();
if (await recommendation) BuyStock(symbol);
```

Interleaving

Consider a case where you're downloading images from the web and processing each image (for example, adding the image to a UI control). You have to do the processing sequentially on the UI thread, but you want to download the images as concurrently as possible. Also, you don't want to hold up adding the images to the UI until they're all downloaded—you want to add them as they complete:

C#

```
List<Task<Bitmap>> imageTasks =
    (from imageUrl in urls select GetBitmapAsync(imageUrl)).ToList();
while(imageTasks.Count > 0)
{
    try
    {
        Task<Bitmap> imageTask = await Task.WhenAny(imageTasks);
        imageTasks.Remove(imageTask);

        Bitmap image = await imageTask;
        panel.AddImage(image);
    }
    catch{}
}
```

You can also apply interleaving to a scenario that involves computationally intensive processing on the ThreadPool of the downloaded images; for example:

C#

```
List<Task<Bitmap>> imageTasks =
    (from imageUrl in urls select GetBitmapAsync(imageUrl)
         .ContinueWith(t => ConvertImage(t.Result)).ToList();
while(imageTasks.Count > 0)
{
    try
    {
        Task<Bitmap> imageTask = await Task.WhenAny(imageTasks);
        imageTasks.Remove(imageTask);

        Bitmap image = await imageTask;
        panel.AddImage(image);
    }
    catch{}
}
```

Throttling

Consider the interleaving example, except that the user is downloading so many images that the downloads have to be throttled; for example, you want only a specific number of downloads to happen concurrently. To achieve this, you can start a subset of the asynchronous operations. As operations complete, you can start additional operations to take their place:

C#

```
const int CONCURRENCY_LEVEL = 15;
```

```
Uri [] urls = …;
int nextIndex = 0;
var imageTasks = new List<Task<Bitmap>>();
while(nextIndex < CONCURRENCY_LEVEL && nextIndex < urls.Length)
{
    imageTasks.Add(GetBitmapAsync(urls[nextIndex]));
    nextIndex++;
}

while(imageTasks.Count > 0)
{
    try
    {
        Task<Bitmap> imageTask = await Task.WhenAny(imageTasks);
        imageTasks.Remove(imageTask);

        Bitmap image = await imageTask;
        panel.AddImage(image);
    }
    catch(Exception exc) { Log(exc); }

    if (nextIndex < urls.Length)
    {
        imageTasks.Add(GetBitmapAsync(urls[nextIndex]));
        nextIndex++;
    }
}
```

Early Bailout

Consider that you're waiting asynchronously for an operation to complete while simultaneously responding to a user's cancellation request (for example, the user clicked a cancel button). The following code illustrates this scenario:

C#

```
private CancellationTokenSource m_cts;

public void btnCancel_Click(object sender, EventArgs e)
{
    if (m_cts != null) m_cts.Cancel();
}

public async  void btnRun_Click(object sender, EventArgs e)
{
    m_cts = new CancellationTokenSource();
    btnRun.Enabled = false;
    try
    {
        Task<Bitmap> imageDownload = GetBitmapAsync(txtUrl.Text);
        await UntilCompletionOrCancellation(imageDownload, m_cts.Token);
```

```
            if (imageDownload.IsCompleted)
            {
                Bitmap image = await imageDownload;
                panel.AddImage(image);
            }
            else imageDownload.ContinueWith(t => Log(t));
        }
        finally { btnRun.Enabled = true; }
    }

    private static async  Task UntilCompletionOrCancellation(
        Task asyncOp, CancellationToken ct)
    {
        var tcs = new TaskCompletionSource<bool>();
        using(ct.Register(() => tcs.TrySetResult(true)))
            await Task.WhenAny(asyncOp, tcs.Task);
        return asyncOp;
    }
```

This implementation re-enables the user interface as soon as you decide to bail out, but doesn't cancel the underlying asynchronous operations. Another alternative would be to cancel the pending operations when you decide to bail out, but not reestablish the user interface until the operations actually complete, potentially due to ending early due to the cancellation request:

C#

```
    private CancellationTokenSource m_cts;

    public async  void btnRun_Click(object sender, EventArgs e)
    {
        m_cts = new CancellationTokenSource();

        btnRun.Enabled = false;
        try
        {
            Task<Bitmap> imageDownload = GetBitmapAsync(txtUrl.Text, m_cts.Token);
            await UntilCompletionOrCancellation(imageDownload, m_cts.Token);
            Bitmap image = await imageDownload;
            panel.AddImage(image);
        }
        catch(OperationCanceledException) {}
        finally { btnRun.Enabled = true; }
    }
```

Another example of early bailout involves using the WhenAny method in conjunction with the Delay method, as discussed in the next section.

Task.Delay

You can use the Task.Delay method to introduce pauses into an asynchronous method's execution. This is useful for many kinds of functionality, including building polling loops and delaying the handling of user input for a predetermined period of time. The Task.Delay method can also be useful in combination with Task.WhenAny for implementing time-outs on awaits.

If a task that's part of a larger asynchronous operation (for example, an ASP.NET web service) takes too long to complete, the overall operation could suffer, especially if it fails to ever complete. For this reason, it's important to be able to time out when waiting on an asynchronous operation. The synchronous Task.Wait, Task.WaitAll, and Task.WaitAny methods accept time-out values, but the corresponding TaskFactory.ContinueWhenAll/Task.WhenAny and the previously mentioned Task.WhenAll/Task.WhenAny methods do not. Instead, you can use Task.Delay and Task.WhenAny in combination to implement a time-out.

For example, in your UI application, let's say that you want to download an image and disable the UI while the image is downloading. However, if the download takes too long, you want to re-enable the UI and discard the download:

C#

```
public async  void btnDownload_Click(object sender, EventArgs e)
{
    btnDownload.Enabled = false;
    try
    {
        Task<Bitmap> download = GetBitmapAsync(url);
        if (download == await Task.WhenAny(download, Task.Delay(3000)))
        {
            Bitmap bmp = await download;
            pictureBox.Image = bmp;
            status.Text = “Downloaded”;
        }
        else
        {
            pictureBox.Image = null;
            status.Text = “Timed out”;
            var ignored = download.ContinueWith(
                t => Trace(“Task finally completed”));
        }
    }
    finally { btnDownload.Enabled = true; }
}
```

The same applies to multiple downloads, because WhenAll returns a task:

C#

```
public async  void btnDownload_Click(object sender, RoutedEventArgs e)
{
    btnDownload.Enabled = false;
    try
    {
```

```
        Task<Bitmap[]> downloads =
            Task.WhenAll(from url in urls select GetBitmapAsync(url));
        if (downloads == await Task.WhenAny(downloads, Task.Delay(3000)))
        {
            foreach(var bmp in downloads) panel.AddImage(bmp);
            status.Text = “Downloaded”;
        }
        else
        {
            status.Text = “Timed out”;
            downloads.ContinueWith(t => Log(t));
        }
    }
    finally { btnDownload.Enabled = true; }
}
```

Building Task-based Combinators

Because a task is able to completely represent an asynchronous operation and provide synchronous and asynchronous capabilities for joining with the operation, retrieving its results, and so on, you can build useful libraries of combinators that compose tasks to build larger patterns. As discussed in the previous section, the .NET Framework includes several built-in combinators, but you can also build your own. The following sections provide several examples of potential combinator methods and types.

RetryOnFault

In many situations, you may want to retry an operation if a previous attempt fails. For synchronous code, you might build a helper method such as `RetryOnFault` in the following example to accomplish this:

C#

```
public static T RetryOnFault<T>(
    Func<T> function, int maxTries)
{
    for(int i=0; i<maxTries; i++)
    {
        try { return function(); }
        catch { if (i == maxTries-1) throw; }
    }
    return default(T);
}
```

You can build an almost identical helper method for asynchronous operations that are implemented with TAP and thus return tasks:

C#

```
public static async  Task<T> RetryOnFault<T>(
    Func<Task<T>> function, int maxTries)
{
    for(int i=0; i<maxTries; i++)
    {
        try { return await function().ConfigureAwait(false); }
        catch { if (i == maxTries-1) throw; }
    }
    return default(T);
}
```

You can then use this combinator to encode retries into the application's logic; for example:

C#

```
// Download the URL, trying up to three times in case of failure
string pageContents = await RetryOnFault(
    () => DownloadStringAsync(url), 3);
```

You could extend the `RetryOnFault` function further. For example, the function could accept another `Func<Task>` that will be invoked between retries to determine when to try the operation again; for example:

C#

```
public static async  Task<T> RetryOnFault<T>(
    Func<Task<T>> function, int maxTries, Func<Task> retryWhen)
{
    for(int i=0; i<maxTries; i++)
    {
        try { return await function().ConfigureAwait(false); }
        catch { if (i == maxTries-1) throw; }
        await retryWhen().ConfigureAwait(false);
    }
    return default(T);
}
```

You could then use the function as follows to wait for a second before retrying the operation:

C#

```
// Download the URL, trying up to three times in case of failure,
// and delaying for a second between retries
string pageContents = await RetryOnFault(
    () => DownloadStringAsync(url), 3, () => Task.Delay(1000));
```

NeedOnlyOne

Sometimes, you can take advantage of redundancy to improve an operation's latency and chances for success. Consider multiple web services that provide stock quotes, but at various times of the day, each service may provide different levels of quality and response times. To deal with these fluctuations, you may issue requests to all the web services, and as soon as you get a response from one, cancel the remaining requests. You can implement a helper function to make it easier to implement this common pattern of launching multiple operations, waiting for any, and then canceling the rest. The `NeedOnlyOne` function in the following example illustrates this scenario:

C#

```
public static async  Task<T> NeedOnlyOne(
    params Func<CancellationToken,Task<T>> [] functions)
{
    var cts = new CancellationTokenSource();
    var tasks = (from function in functions
                 select function(cts.Token)).ToArray();
    var completed = await Task.WhenAny(tasks).ConfigureAwait(false);
    cts.Cancel();
    foreach(var task in tasks)
    {
        var ignored = task.ContinueWith(
            t => Log(t), TaskContinuationOptions.OnlyOnFaulted);
    }
    return completed;
}
```

You can then use this function as follows:

C#

```
double currentPrice = await NeedOnlyOne(
    ct => GetCurrentPriceFromServer1Async("msft", ct),
    ct => GetCurrentPriceFromServer2Async("msft", ct),
    ct => GetCurrentPriceFromServer3Async("msft", ct));
```

Interleaved Operations

There is a potential performance problem with using the WhenAny method to support an interleaving scenario when you're working with very large sets of tasks. Every call to WhenAny results in a continuation being registered with each task. For N number of tasks, this results in O(N2) continuations created over the lifetime of the interleaving operation. If you're working with a large set of tasks, you can use a combinator (`Interleaved` in the following example) to address the performance issue:

C#

```
static IEnumerable<Task<T>> Interleaved<T>(IEnumerable<Task<T>> tasks)
{
    var inputTasks = tasks.ToList();
    var sources = (from _ in Enumerable.Range(0, inputTasks.Count)
                   select new TaskCompletionSource<T>()).ToList();
    int nextTaskIndex = -1;
```

```
    foreach (var inputTask in inputTasks)
    {
        inputTask.ContinueWith(completed =>
        {
            var source = sources[Interlocked.Increment(ref nextTaskIndex)];
            if (completed.IsFaulted)
                source.TrySetException(completed.Exception.InnerExceptions);
            else if (completed.IsCanceled)
                source.TrySetCanceled();
            else
                source.TrySetResult(completed.Result);
        }, CancellationToken.None,
           TaskContinuationOptions.ExecuteSynchronously,
           TaskScheduler.Default);
    }
    return from source in sources
           select source.Task;
}
```

You can then use the combinator to process the results of tasks as they complete; for example:

C#

```
IEnumerable<Task<int>> tasks = ...;
foreach(var task in Interleaved(tasks))
{
    int result = await task;
    …
}
```

WhenAllOrFirstException

In certain scatter/gather scenarios, you might want to wait for all tasks in a set, unless one of them faults, in which case you want to stop waiting as soon as the exception occurs. You can accomplish that with a combinator method such as `WhenAllOrFirstException` in the following example:

C#

```
public static Task<T[]> WhenAllOrFirstException<T>(IEnumerable<Task<T>> tasks)
{
    var inputs = tasks.ToList();
    var ce = new CountdownEvent(inputs.Count);
    var tcs = new TaskCompletionSource<T[]>();

    Action<Task> onCompleted = (Task completed) =>
    {
        if (completed.IsFaulted)
            tcs.TrySetException(completed.Exception.InnerExceptions);
        if (ce.Signal() && !tcs.Task.IsCompleted)
            tcs.TrySetResult(inputs.Select(t => t.Result).ToArray());
```

```
        };

        foreach (var t in inputs) t.ContinueWith(onCompleted);
        return tcs.Task;
    }
```

Building Task-based Data Structures

In addition to the ability to build custom task-based combinators, having a data structure in Task and Task(Of TResult) that represents both the results of an asynchronous operation and the necessary synchronization to join with it makes it a very powerful type on which to build custom data structures to be used in asynchronous scenarios.

AsyncCache

One important aspect of a task is that it may be handed out to multiple consumers, all of whom may await it, register continuations with it, get its result or exceptions (in the case of Task(Of TResult)), and so on. This makes Task and Task(Of TResult) perfectly suited to be used in an asynchronous caching infrastructure. Here's an example of a small but powerful asynchronous cache built on top of Task(Of TResult):

C#

```
public class AsyncCache<TKey, TValue>
{
    private readonly Func<TKey, Task<TValue>> _valueFactory;
    private readonly ConcurrentDictionary<TKey, Lazy<Task<TValue>>> _map;

    public AsyncCache(Func<TKey, Task<TValue>> valueFactory)
    {
        if (valueFactory == null) throw new ArgumentNullException("loader");
        _valueFactory = valueFactory;
        _map = new ConcurrentDictionary<TKey, Lazy<Task<TValue>>>();
    }

    public Task<TValue> this[TKey key]
    {
        get
        {
            if (key == null) throw new ArgumentNullException("key");
            return _map.GetOrAdd(key, toAdd =>
                new Lazy<Task<TValue>>(() => _valueFactory(toAdd))).Value;
        }
    }
}
```

The AsyncCache<TKey,TValue> class accepts as a delegate to its constructor a function that takes a **TKey** and returns a Task(Of TResult). Any previously accessed values from the cache are stored in the internal dictionary, and the

AsyncCache ensures that only one task is generated per key, even if the cache is accessed concurrently.

For example, you can build a cache for downloaded web pages:

C#

```
private AsyncCache<string,string> m_webPages =
    new AsyncCache<string,string>(DownloadStringAsync);
```

You can then use this cache in asynchronous methods whenever you need the contents of a web page. The **AsyncCache** class ensures that you're downloading as few pages as possible, and caches the results.

C#

```
private async  void btnDownload_Click(object sender, RoutedEventArgs e)
{
    btnDownload.IsEnabled = false;
    try
    {
        txtContents.Text = await m_webPages["http://www.microsoft.com"];
    }
    finally { btnDownload.IsEnabled = true; }
}
```

AsyncProducerConsumerCollection

You can also use tasks to build data structures for coordinating asynchronous activities. Consider one of the classic parallel design patterns: producer/consumer. In this pattern, producers generate data that is consumed by consumers, and the producers and consumers may run in parallel. For example, the consumer processes item 1, which was previously generated by a producer who is now producing item 2. For the producer/consumer pattern, you invariably need some data structure to store the work created by producers so that the consumers may be notified of new data and find it when available.

Here's a simple data structure built on top of tasks that enables asynchronous methods to be used as producers and consumers:

C#

```
public class AsyncProducerConsumerCollection<T>
{
    private readonly Queue<T> m_collection = new Queue<T>();
    private readonly Queue<TaskCompletionSource<T>> m_waiting =
        new Queue<TaskCompletionSource<T>>();

    public void Add(T item)
    {
        TaskCompletionSource<T> tcs = null;
        lock (m_collection)
        {
```

```
            if (m_waiting.Count > 0) tcs = m_waiting.Dequeue();
            else m_collection.Enqueue(item);
        }
        if (tcs != null) tcs.TrySetResult(item);
    }

    public Task<T> Take()
    {
        lock (m_collection)
        {
            if (m_collection.Count > 0)
            {
                return Task.FromResult(m_collection.Dequeue());
            }
            else
            {
                var tcs = new TaskCompletionSource<T>();
                m_waiting.Enqueue(tcs);
                return tcs.Task;
            }
        }
    }
}
```

With that data structure in place, you can write code such as the following:

C#

```
private static AsyncProducerConsumerCollection<int> m_data = …;
…
private static async  Task ConsumerAsync()
{
    while(true)
    {
        int nextItem = await m_data.Take();
        ProcessNextItem(nextItem);
    }
}
…
private static void Produce(int data)
{
    m_data.Add(data);
}
```

The System.Threading.Tasks.Dataflow namespace includes the BufferBlock(Of T) type, which you can use in a similar manner, but without having to build a custom collection type:

C#

```
private static BufferBlock<int> m_data = …;
…
```

```
private static async  Task ConsumerAsync()
{
    while(true)
    {
        int nextItem = await m_data.ReceiveAsync();
        ProcessNextItem(nextItem);
    }
}
…
private static void Produce(int data)
{
    m_data.Post(data);
}
```

Note

The System.Threading.Tasks.Dataflow namespace is available in the .NET Framework 4.5 through **NuGet**. To install the assembly that contains the System.Threading.Tasks.Dataflow namespace, open your project in Visual Studio 2012, choose **Manage NuGet Packages** from the Project menu, and search online for the Microsoft.Tpl.Dataflow package.

See Also

Task-based Asynchronous Pattern (TAP)
Implementing the Task-based Asynchronous Pattern
Interop with Other Asynchronous Patterns and Types